Proposed principles: Reflect C++, generate C++ (by default)

Document Number: **P3437 R1**
Date: 2024-10-31
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: SG7, EWG

Contents

Abstract

This paper proposes that we adopt principles to guide reflection and generation proposals. Specifically:

- human programmers read and write source code, and want to (and can most easily successfully reason about) `consteval` code that automates what they would otherwise do by hand;

and therefore

- reflection should reflect C++ source code (by default)
- generation should generate C++ source code (by default)

1 Motivation

The reflection and generation papers are converging very well in mostly good directions.

However, it is important to be conscious of principles in order to reduce design risk.

This paper proposes a simple principle: That reflection and generation should be **of C++ language source code**, at least by default. Phrasing it this way is intended to allow for other designs (e.g., generation that bypasses access control) as long as those are not the language default (i.e., not easy to spell by accident because it's the "nicest" syntax; unsafe operations should be visible and require an "uglier" spelling).

1.1 Terms

In this paper:

- ***Reflection*** means getting meta-information about the program, such as the number of data members of a type. (Sometimes other papers use words like "introspection" as synonyms.)
- ***Generation*** means creating new parts of the program, such as adding a new class or expression. (Sometimes other papers use words like "injection" as synonyms.)
- ***Language source code*** (for short, ***source code***) means the C++ language source code, excluding the textual preprocessor.

2 Observation: Programmers read and write source code; source code is "the UI of C++"

In R0 of this paper I focused on the metaphor that the source code is the Source of Truth, as Daveed Vandevoorde put it. I agreed (and still agree) with that, but that metaphor could be misunderstood here because the parse tree or fully bound tree is sometimes also the "source of truth" in important ways.

So in R1 of this paper I'm instead focusing on the metaphor that the source code is the UI of C++. It's what programmers read to understand the program, and it's what programmers write to create and add to the program. Importantly, it is what programmers are best at successfully reasoning about (despite C++ source's complexities and arcana; programmers are used to thinking in code by default, not in ASTs by default).

3 Proposal: Reflection and generation should automate that (by default)

3.1 Reflection should reflect C++ source code (the whole language)

Principle: A primary use case of reflection is as a proxy for the human code reader. A primary use case (and default programmer expectation) is that reflection will enable programmers to write compile-time code that can "read" the same things as the human programmer, so that the programmer can write compile-time code to use that meta-information to automate doing things in code instead of manually by the human programmer.

Corollary: By default, reflection should reflect source code, as if read by eye.

Corollary: All information in the program source code that the human reader can see or infer must also be reflectable (eventually, even if successive proposals gradually add the ability to reflect more and more of the program). Why: If the programmer can know it from reading the code, they'll want to use it.

Example: Attributes. In the past there have been discussions on whether attributes would ever need to be reflectable. This principle answers that question without need for discussion: Yes, of course, attributes must be reflectable (eventually, even if not in an initial reflection proposal), because the human programmer can see them and so will want to write meta-programs to automate things that otherwise the human programmers would have to do manually.

Example: Default accessibility. Consider this type body:

```
{
    int f();
};
```

Because there is no access-specifier before `f`, the human programmer can look at this type definition body and know that `f` has *default* accessibility. Therefore, if this is the body of a `class` then `f` is private, and if this is the body of a `struct` then `f` is public. Therefore the human programmer will need to be able to query this information in a meta-program ("does this member declaration appear before an access-specifier?"); [P0707R4] and [cppfront] are full of examples where this is important, because a metafunction is being used to assign a "default" accessibility only if the human programmer did not write an explicit access-specifier.

3.2 Generation should generate C++ source code (the whole language; not a C++ dialect or divergent language)

Principle: A primary use case of generation is as a proxy for the human code writer. A primary use case (and default programmer expectation) is that generation will enable programmers to write compile-time code that can "write" the same things as the human programmer, so that the programmer can write compile-time code to use that meta-information to automate writing code that would otherwise be written manually by the human programmer.

Corollary: By default, generation should generate source code, as if written by hand.

Corollary: All information in the program source code that the human reader can write must also be generatable (eventually, even if successive proposals gradually add the ability to generate more and more of the language). Why: If the programmer can write it by hand, they'll want to automate generating it programmatically.

Corollary: By default, that means ordinary C++ semantics. By default, generated code should have the same meaning as if it had been written by hand.

Example: [P3294R1] token injection. As proposed in [P3294R1], token injection does follow this principle — as long as injecting a token sequence is always identical in meaning to injecting a source code string and then tokenizing it normally to get the identical token sequence (i.e., it should be equivalent to textual language source code).

Example: [P2996R5] splices. I view splices as a narrow form of generation. However, because splices are currently specified to bypass access checking, they are not the same as generating source code; for example, `obj.x` and `obj.[:reflect_x:]` do not have the same semantics. Therefore this principle argues that they should be

provided under a more-different non-default syntax (e.g., perhaps one that directly looks like what we already have: pointers to members, if that is the splice mental model), because they are not generating the same semantics the programmer would write by hand. Providing them under a "nice" syntax will be a source of surprise.

Example: Language features available only in generated code should be avoided. I have heard some suggestions that generated code be able to generate constructs that the programmer cannot otherwise write by hand (e.g., an "overload set" abstraction available only in generated code, but not handwritten code). But that means that generated code is a an extended dialect of C++ (or in the worst case something completely different), not ordinary C++. And if such features are desirable, they'll be wanted not only in generated code, but in handwritten code too. — No doubt we will learn about such new features that are useful, and that's fine; but we should always separate them out as orthogonal language extensions with their own P-numbered proposals for the ordinary handwritten language; if they are available to ordinary hand-written code, generated code can also naturally produce them. They should not be added to generated code only "while we're at it" to add language features in generated code's clothing.

Corollary: Generation should be able to generate arbitrary local output. Generated code may need to be saved as text to disk (e.g., as a text `.h` file) that other translation units can share, and not be limited to use only within the current translation unit. Generated code may need to include generating binary output files, such as output files for other build tools (e.g., for WinRT interoperability and to replace the IDL "side compilers," generation must be able to generate `.winmd` output files).

3.3 Additional rationale: Tooling needs source code

Principle: All source code must be visible, whether hand-written or generated. The final code is the only source of truth, regardless whether it is handwritten or generated. We must be able to see the program we are compiling.

The basic requirement is that generated code should be pretty-printable to make it visible.

More advanced requirements include tooling: A debugger must be able to step into generated code. An IDE must be able to visualize generated code (e.g., expand/collapse from the handwritten/full code). Consider that we have always had such cases of generated code because of hardwired language semantics, such as the automatically generated special member functions: There has always been a demand to be able to do things like "step into a generated copy constructor." That same requirement is now just being generalized.

4 References

[cppfront] H. Sutter. Cppfront compiler (GitHub, 2022-2024).

[P0707R4] H. Sutter. "Metaclass functions: Generative C++" (WG21 paper, June 2019).

[P2996R5] W. Childers, P. Dimov, D. Katz, B. Revzin, A. Sutton, F. Vali, D. Vandevoorde. "Reflection for C++26" (WG21 paper, August 2024).

[P3294R1] A. Alexandrescu, B. Revzin, D. Vandevoorde. "Code injection with token sequences" (WG21 paper, July 2024).